第9章 进 程

一路走来辛辛苦苦，现在终于到进程这一步了。

进程管理、内存管理和文件系统管理是操作系统的三个最重要的功能。这其中，进程的重要程度非同一般。关于内存管理和文件系统管理两部分内容，前面章节中已介绍过了，本章我们将会揭开进程的面纱，一个操作系统的全貌将会最终呈现。

9.1 进程的实现原理

什么是进程呢？这是一个常识性的问题。通俗地讲，进程就是一个执行中的程序。一个应用程序放在存储设备中不用，它只能叫做程序。而当我们把这个程序读取到内存中并运行时，一个进程就产生了。

应用程序往往不是一个挨着一个去运行的，例如，一个人在浏览网页的同时，也在欣赏着美妙的音乐，可能还要时不时地跟在线的朋友聊上两句，无论是聊天程序、音乐播放程序还是网页浏览程序，只要它们都处在运行的状态，那么这些进程就都是同步的。

除了并行执行外，进程通常还具有另外一个特点，那就是每个进程的地址空间都是独立的，比如，一个聊天的程序没有办法随意修改网页浏览程序的变量，除非它们通过某种方法预先进行了沟通。

以上两个方面描述的其实是进程模型的两个基本属性，总结成一句话，就是从执行逻辑上看，进程是并行执行的，从内存空间上看，进程则是彼此独立的。

话又说回来，进程模型的这两个基本属性只是针对多任务操作系统来说的。对于那些单任务的操作系统，情况可就不一样了。DOS 是一个典型的单任务操作系统。想要在 DOS 系统中运行多个程序，只能是先运行一个，等到这个程序执行完毕，才能运行另一个。在这样的操作系统中，进程不存在并行的问题。

在了解了进程的一般特点后，我们就不得不去研究一个具体的问题了。这涉及到我们的操作系统进程部分的顺利实现。要知道一切硬件平台的 CPU 永远都是有限多个，而操作系统理论上却能够并行运行无限多个进程。怎样做到让有限个 CPU 同时执行无限多个程序呢？

这个问题其实只有一个解决办法，那就是宏观并行、微观串行。我们让每一个要执行的进程都只执行一段很短的时间，如 1 毫秒。在这段时间用完之后，不管这个进程是否执行完成，CPU 都将放弃执行这个进程，而选择别的进程去运行。这样，虽然从微观上看我们的进程依然是顺序执行的，但从宏观上审视运行的各个进程时，进程就变成并行执行的了。

这种思想是实现进程并行执行的最基本的方法。其中，让每一进程都执行一小段的这段时间，专业术语叫做“时间片”。而当一个进程的时间片耗尽，CPU 选择下一个进程去执行的这个动作，就叫做进程切换，整个过程可以用图 9-1 来描述。

接下来我们需要考虑的问题就是，一个进程怎么就可以在没有执行完的情况下，中途切换到别的进程去执行呢？寄希望于进程本身并不现实。这其实是说让系统中每一个进程在运行时，都要定时检查自己的运行时间是否超过了某个时间片，如果超过了运行时间片，就立刻放弃执行，这是绝对不可能的。

既然进程自身不可能定时地去检测超时，那么我们就让操作系统去做这个工作。不要忘了，几乎是所有的 CPU 都会通过内部或外部的方式提供某种定时机制。当某一特定时间到来时，硬件会触发一个中断，那么在中断处理程序中，系统将迫使 CPU 放弃执行一个进程，选择一个新的进程去运行。于是我们就得出了实现进程的一个非常重要的结论：依靠硬件时钟中断在中断处理程序中实现进程切换。

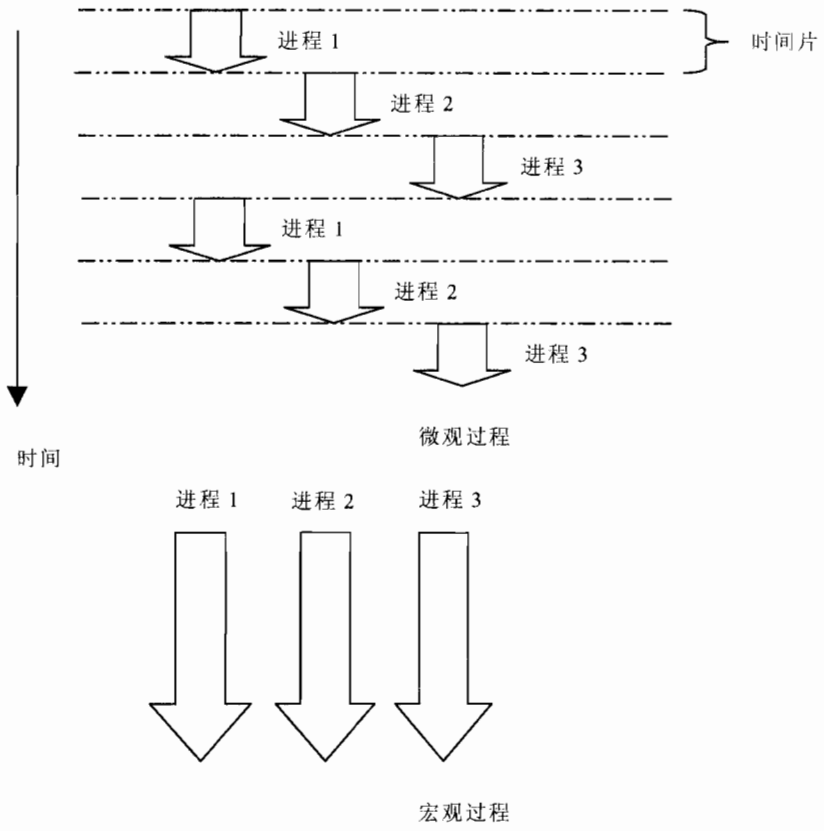


图 9-1 进程调度过程

在时钟中断处理程序中，系统需要做哪些工作才能保证进程顺利切换呢？读者朋友们应该都会很自然地想到，只要记录上一个进程切换时正在运行的那条代码的地址，同时将下一个要运行的进程所保存的地址读出来，再跳转过去运行，不就可以了吗？

当然就是这个意思，但过程却没有这么简单。进程在切换时，被终止的那条指令地址当然是必须保存的，除此之外，还包括所有寄存器、程序状态、程序堆栈、进程的地址空间……这个过程，叫做保护现场。于是，我们需要为每一个进程预留出一块内存空间，专门用于保存进程切换时的现场。很显然，这块内存区应该尽可能独立，不能让别的进程轻易地访问到。

另一个棘手的问题是，既然进程与进程之间需要完全隔离开，那么这就意味着每个进程所使用的堆栈必须彼此独立，这样才能保证进程在执行时，

彼此之间没有干扰。

还有，进程本身非常复杂、属性很多，需要使用自定义的数据类型来描述。在 C 语言中，结构体就是专门用来描述自定义类型的。于是通常的做法是定义一个结构体来描述每一个进程。当一个新的进程产生时，系统就需要为该进程分配一个结构体。

以上三个问题，本质上来讲，还是内存分配的问题。当一个新的进程产生时，系统就给它分配三块独有的内存区，一块用于存储进程的运行环境，另一块作为进程堆栈，还有一块用来存储描述进程的结构体。

其实我们完全可以在进程产生时，只给它分配一块空间，而让上述三种内存需求都在这一块内存空间中完成。这既是对程序编码的一种简化，也是对进程执行效率的一种提升。这种对进程的处理方法，一定程度上是源自 Linux 的。我们不妨先来了解一下 Linux 是怎么解决这个问题的。

在 Linux 系统中，当一个进程产生时，内核就会给该进程分配一个专有内存区。将这个专有内存区的一端作为堆栈首地址，而将描述进程的结构体保存到内存的另一端中。这段内存空间的大小在 Linux 中通常是 8K。当然，这个值绝不是 Linux 一拍脑门想到的。选择 8K 的原因是它恰好等于两个页的大小。这样，在内存分配时，系统就可以直接给进程分配一个 order 为 1 的两个页的空间，从而可以迅速获得内存。当然，这里也可以根据实际内存情况适当地选择 4K、16K 或其他尺寸。

同时，系统对这段内存空间还有一个硬性要求，那就是内存必须按照一个页的整数被对齐。举个例子来说，如果一个内存空间位于 0x30211000～0x30212000，那么，对于进程来说就是可用的，但如果内存的位置是 0x30325020～0x30326020，则该内存不能被进程使用。这是 Linux 的一个巧妙设计，原因我们稍后会详述。

Linux 将用于描述进程信息的结构体命名为 thread_info。于是，对于一个堆栈向下生长的体系结构来说，Linux 下一个进程的内存空间类似于图 9-2 所描述的那样。

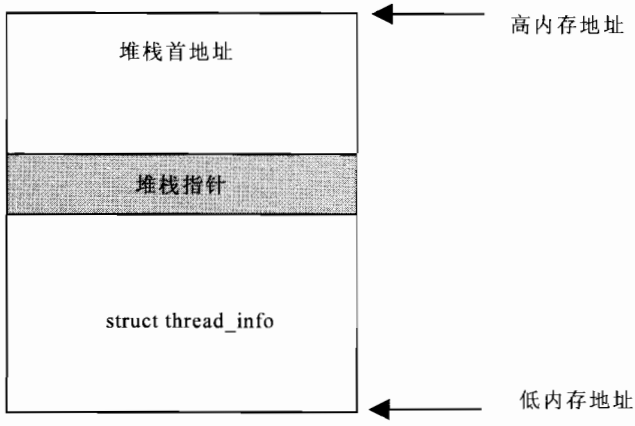


图 9-2 Linux 的进程内存

有的读者朋友可能会觉得这种设计似乎没有什么特别之处。但如果我们需要获得描述当前进程的结构体时，这可能就是最快的办法了。不要忘了我们前面说过，分配给进程的这个 8K 内存空间是按照页的整数倍对齐的。这样，无论当前堆栈指针指向何处，我们都可以让它和 0xffffe000 做“与”操作，得到的结果将必然指向这段内存的低地址，也就是结构体 struct thread_info 的地址。

在 x86 体系下，这段代码的实现如下：

代码 9-1

```
movl $-8192,%eax
andl %esp,%eax
```

而在 ARM 体系结构中，获取结构体的地址可以使用下面这样的方式来完成：

代码 9-2

```
struct task_info *current_task_info(void){
    register unsigned long sp asm ("sp");
    return (struct task_info *)(sp&~(8192-1));
}
```

从这两段代码中我们可以看出，取出堆栈指针寄存器的值并通过“与”的操作来得到进程结构体的首地址，无论在哪种硬件平台下都是很方便的。

这样，在操作系统中实现进程的所有理论问题，我们都解释清楚了。事

实证明这些理论并不难理解。对操作系统原理稍懂一些的读者都知道进程切换要靠时钟中断来驱动，也知道进程在切换时需要保护现场。但落实到代码中，恐怕绝大多数人是会手足无措的。毕竟进程调度的实现逻辑不同于普通程序，是需要很多技巧的。

事不宜迟，下面就让我们从实践的角度出发，让进程也能在我们的操作系统中运行。

9.2 进程的实现

首先我们需要改写一下系统的中断处理方式，让原有的时钟中断能够负担起进程调度的责任。

9.2.1 改写中断处理程序

为了让程序看起来更清晰，我们删除了原来的中断处理程序，只保留了与进程切换有关的部分。

代码 9-3

```
__vector_irq:
    sub r14,r14,#4
    stmfd r13!,{r0}
    stmfd r13!,{r1-r3}

    mov r2,#0xca000000
    add r1,r2,#0x10
    ldr r0,[r1]
    ldr r3,[r2]
    orr r3,r3,r1
    str r3,[r2]
    str r0,[r1]

    ldmfd r13!,{r1-r3}
    mov r0,r14
    CHANGE_TO_SYS
```

```
    stmfd r13!,{r0}
    stmfd r13!,{r14}
    CHANGE_TO_IRQ
    ldmfd r13!,{r0}
    ldr r14,=__asm_schedule
    stmfd r13!,{r14}
    ldmfd r13!,{pc}^

__asm_schedule:
    stmfd r13!,{r0-r12}
    mrs r1, cpsr
    stmfd r13!,{r1}

    mov r1,sp
    bic r1,#0xff0
    bic r1,#0xf
    mov r0,sp
    str r0,[r1]

    bl __common_schedule
    ldr sp,[r0]
    ldmfd r13!,{r1}
    msr cpsr,r1
    ldmfd r13!,{r0-r12,r14,pc}
```

代码 9-3 就是这样一段程序。首先，程序通过一条 sub 指令修正了返回时的地址。然后，将寄存器 R0 ~ R3 压入中断模式下的堆栈。虽然这段程序是针对进程切换而设计的，但我们同样可以看到，这些处理方法与通用的中断处理程序没有什么差别。

在接下来的一段代码中，我们需要做的就是清除掉 s3c2410 中断控制器与时钟有关的寄存器，避免中断不停产生。中断控制器的物理地址本应是从 0x4a000000 开始的，但由于我们的操作系统已经开启了 MMU，于是程序必须通过虚拟地址 0xca000000 作为基地址对控制器进行访问。如果读者对这段代码依旧似懂非懂，可以翻看前面中断一章的内容，那里有非常详细的讲解。

之后的这段代码便与进程调度有关了。程序首先把寄存器 R1 ~ R3 从堆栈中恢复出来。这里没有将寄存器 R0 弹出堆栈的原因是，我们需要通过 R0 来实现不同异常模式间的参数传递。于是在中断模式下，程序通过 mov

指令将寄存器 R14 的值保存到 R0 里。这也就意味着，上一个进程的返回地址此时被保存到了寄存器 R0 中。紧接着，程序通过 CHANGE_TO_SYS 宏切换到系统模式下，又将寄存器 R0 和系统模式下的 R14 压入了系统模式的堆栈之中。

读者可以仔细品味一下这种操作方法的真正用意，R0 是中断发生时上一个状态的返回值，但也可以说成是上一个进程被切换时将要执行的指令地址，而寄存器 R14 保存的则是上一个进程在某个函数调用时的返回地址。将上一个进程的这两个值压入的这块堆栈区，除了可以理解成是系统模式下的堆栈外，还可以理解成是上一个进程的堆栈，这只需要通过改写 R13 寄存器的值就可以实现。

于是我们得到了一个非常重要的结论：当时钟中断发生时，中断处理程序负责进程调度，首先要将上一个进程的要执行的那条指令的地址和函数返回值地址压入上一个进程的私有堆栈中，如图 9-3 所示。

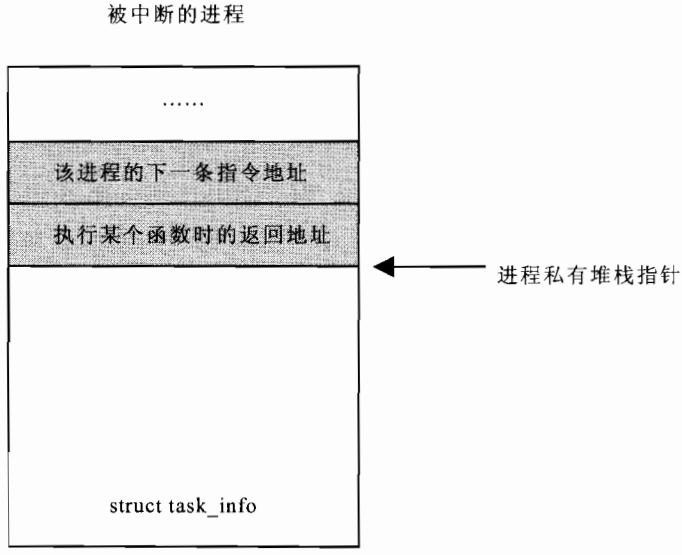


图 9-3　进程内存空间

在完成了上述操作之后，程序通过 CHANGE_TO_IRQ 宏返回到中断模式，然后从中断模式下的堆栈中恢复寄存器 R0 的值。到这一步为止，程序好像是什么都没有发生一样，又恢复到了刚刚发生中断时的样子。殊不知这

是“明修栈道，暗渡陈仓”，程序悄悄地从中断模式切换到系统模式，将两个关键值压栈，又悄悄地返回到了中断模式中。

之后的动作就有些巧妙了，函数__asm_schedule 的地址被赋值给了 R14 寄存器，然后将该寄存器压入堆栈，紧接着又将它弹出堆栈。所不同的是，此时堆栈中的值不是恢复到寄存器 R14 里，而是直接恢复到程序计数器 PC 中，同时也将寄存器 SPSR 的值恢复到 CPSR 中去。

这个过程从本质上讲就是一个程序的跳转，只不过跳转的同时要伴有模式的切换。为什么会选择寄存器 R14 来实现这个过程呢？原因是中断模式下的 R14 是唯一一个空闲不用的寄存器。使用这种独特的跳转方法的另一个目的是，这样可以实现在程序跳转的同时更改运行模式。

于是，程序恢复到了中断之前的模式中，或者更贴切地说，是上一个进程所处的模式之中，与此同时，程序跳转到了函数__asm_schedule 处开始运行。

这样，一段原本要从一个进程切换到另一个进程的程序似乎又回到了原来的进程之中了。

这种观点当然是错误的，此时虽然程序回归到原来的进程当中，但根本目的不是继续原来的进程运行，而是在原进程环境下保护进程现场。__asm_schedule 函数的核心功能就在于此。

在__asm_schedule 函数中，程序首先将原进程的 R0～R12 寄存器都压入到了该进程的堆栈之中，之后又借助寄存器 R1 将程序状态寄存器的值也压入堆栈。接下来，我们又将堆栈指针寄存器 SP 赋值给了 R1，然后通过两次 BIC 指令清除掉 R1 寄存器的后 12 位。

朋友们是否已经意识到了这段程序的真正用意了？没错，此时 R1 寄存器的值恰好是进程所属内存区的低地址。回头看一下图 9-2，大家就会完全明白了。最终，寄存器 R1 的值等于该进程结构体的地址。只不过我们没有效仿 Linux，将这个结构体命名为 thread_info，而是称之为 task_info。该进程的当前堆栈指针就保存在这个地址中。如果读者还在疑惑，为什么一个结构体的地址可以用来保存堆栈指针的值，那么就请看一下 struct task_info 结构体的定义吧。

代码 9-4

```
struct task_info{
    unsigned int sp;
```

```
    struct task_info *next;
};
```

通过代码 9-4 我们知道，struct task_info 结构体的第一个成员是一个无符号整型变量，名字叫做 sp，这个变量就是专门用来保存进程切换时的堆栈指针的。

这样，进程保护现场的工作就成功完成了，完成之后的进程堆栈如图 9-4 所示。

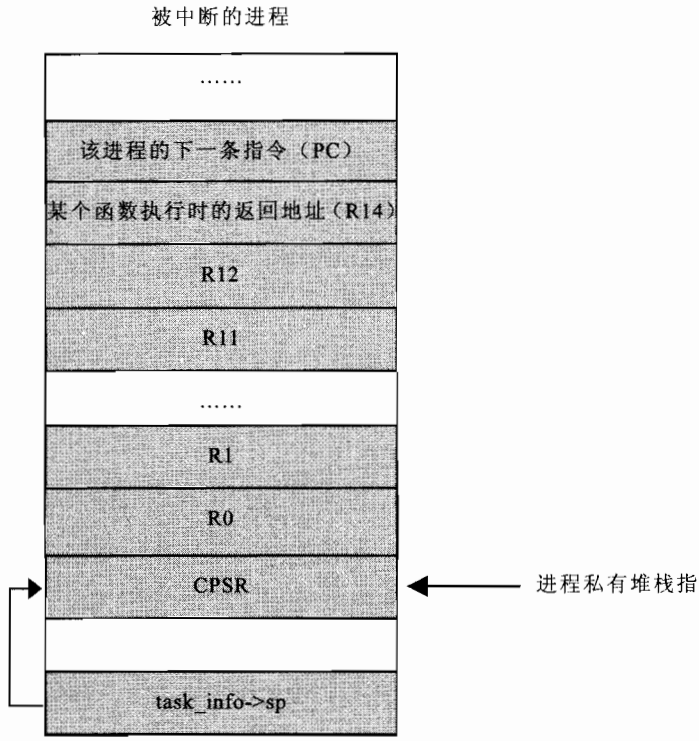


图 9-4　进程私有堆栈空间

9.2.2　抽象调度函数

虽然进程保护现场的过程结束了，但进程切换过程远没有结束。所以在代码 9-3 中，程序紧接着调用了一个名为__common_schedule 的函数。这个函数与硬件平台无关，它并不需要在指定的硬件平台下运行。因此我们将该函数定义到了别处，并使用 C 语言去实现，这就是所谓的抽象调度函数。

抽象调度函数能够返回下一个进程的 struct task_info 结构体首地址，也就是下一个进程所属的内存空间的低地址。不要忘了，进程在切换时会将所有资源都保存到这段空间之中。于是，在得到这段内存空间的基地址之后，我们就可以得到相应的进程资源，并将它们全部恢复。这样，进程切换就算完成了。

通过上面的描述，读者应该可以体会到这种抽象的真正含义，第一，这个函数本身与硬件无关，理论上独立于任何硬件体系结构之上；第二，只要这个函数的返回值不变，无论其实现方法是怎样的，都能够确保进程切换的正确性。

正是凭借这种抽象性，使我们能在今后的操作系统优化过程中更加容易地实现各种不同的进程调度算法，从这个角度来看，该函数的意义非常重大。

进程调度算法对于操作系统来说是非常重要的，简单说来，进程调度算法是用来决定某个进程是运行的时间长一点还是短一点的算法，它会直接影响到操作系统的运行效率。然而，本书将不会对进程调度算法进行深入研究，因为进程的问题一旦深入，就会引出源源不断的新话题，这不是本书的初衷。因此，在这里我们会采用一种最简单的进程调度算法来实现进程的切换，那就是基于时间片的轮询调度算法。

说白了，这种方法就是让所有待运行的进程挨个排队，每个进程运行一段固定的时间，循环往复直至程序结束。

轮询调度算法简单到什么程度，看一下__common_schedule 函数的具体实现就清楚了。

代码 9-5

```
void *__common_schedule(void){
    return (void *)(current->next);
}
```

代码 9-5 仅仅将当前 current 的 next 成员返回。这里，current 代表的正是当前正在运行的进程，具体的定义方法我们稍后会介绍。我们知道，系统中的所有进程会在初始化时被连接成一个环形链表，每次取得当前进程的 next 成员就意味着程序在循环遍历链表中的所有进程，从而实现对进程的轮询调度。

一旦__common_schedule 函数运行完成，寄存器 R0 的值就是下一个进

程的 struct task_info 结构体的地址了。并且根据前面的描述我们知道，这个进程在切换之前，也会将该进程的私有堆栈指针保存到 struct task_info 结构体的 sp 成员处。于是在代码 9-3 中，程序通过一条 ldr 指令成功地恢复了 R13 寄存器。

接下来，程序又将保存在堆栈中的 CPSR 寄存器的值恢复。最后将除 R13 寄存器外的所有其他寄存器的值依次恢复。当然，这其中也包括程序计数器 PC，而恢复给 PC 寄存器的正是进程被切换时恰要运行的那条指令。

至此，进程切换的所有工作就完成了。

9.2.3 新进程的产生

代码 9-3 为我们展现了多进程同时运行的一种工作方法。现在的问题是，系统从启动开始就是以单进程的形式出现的，我们应该怎样产生一个新进程并与原有的进程并行运行呢？

一些简单的操作系统在实现多进程时，会采用静态或半静态的方式。这种方法的实现思路是，要在这种系统中运行的进程，其功能和数量都是固定的，也就是说，用户进程的信息需要被写入到操作系统源代码中，每次当我们需要更改用户进程时，就必须重新编译系统。在这样的操作系统中，开新进程的方法虽然简单，但同时也带来一个问题，即操作系统与应用程序之间的耦合太紧密了，系统过分限制了用户，没能留给用户一个自由的环境去任意发挥。

所以在多数高级操作系统中，开新进程的工作往往会由函数来实现。这样，当一个用户想要以多进程的方式运行程序时，只需要调用系统提供的函数就可以了。熟悉 Linux 的朋友应该都会想到，Linux 下的 fork 系统调用就是这样一个函数。

考虑到操作系统的设计初衷，在我们的操作系统中，不妨也将开新进程的动作封装到一个函数之中来实现动态的进程管理。

如果读者理解了代码 9-3 的工作原理，那么开一个新进程的工作将不难实现。图 9-5 描述了生成一个新进程的四个基本步骤。

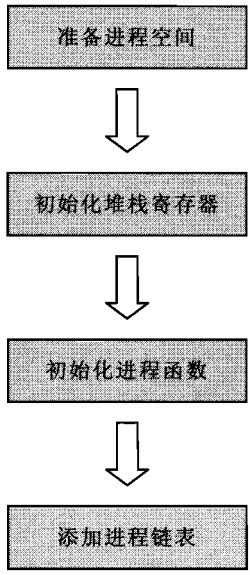


图 9-5 新进程产生的步骤

首先，我们需要为新进程开辟出一块内存空间，用来保存进程运行时需要的所有数据。结合图 9-3 我们知道，struct task_info 结构体就位于该空间的低地址端。

然后，我们要初始化新进程的堆栈寄存器。在这里，初始化寄存器指的并不是要真地去改写堆栈寄存器的值，而只是要初始化结构体 struct task_info 中 sp 成员的值。这样，一旦这个进程被调度，系统就会读取该值并赋值给寄存器 SP，就像代码 9-3 那样。那这个值应该是多少才正确呢？读者只需要参考一下图 9-4，就会知道答案。

接下来的工作是初始化进程函数。进程是一段执行着的程序，本质上还是一段代码。那么，当一个新的进程产生时，如何让它去执行既定代码？恐怕最简单的方法莫过于函数。我们把新进程将要完成的动作都写到一个函数中，然后将这个进程的入口地址压入进程堆栈中，等到进程切换时，系统将该地址恢复，于是该函数将以一个全新进程的方式去运行。

最后我们还需要将与新进程有关的数据信息保存起来。这样，当进程切换时，__common_schedule 函数就能够通过某种方法找到并返回该进程的 struct task_info 结构体，完成进程切换了。通常，系统都会选择链表这种方

式来保存进程信息，理论上这样可以支持任意多的进程同时运行。

将上述过程转换成代码，我们会得到：

代码 9-6

```
#define disable_schedule(x) disable_irq()
#define enable_schedule(x)  enable_irq()

int task_stack_base=0x30300000;
struct task_info *copy_task_info(struct task_info *tsk){
    struct task_info *tmp=(struct task_info *)task_stack_base;
    task_stack_base+=TASK_SIZE;
    return tmp;
}

#define DO_INIT_SP(sp,fn,args,lr,cpsr,pt_base)     \
    do{\
        (sp)=(sp)-4;/*r15*/     \
        *(volatile unsigned int *)(sp)=(unsigned int)
(fn);/*r15*/ \
        (sp)=(sp)-4;/*r14*/     \
        *(volatile unsigned int *)(sp)=(unsigned int)(lr);/*r14*/\
        (sp)=(sp)-4*13;/*r12,r11,r10,r9,r8,r7,r6,r5,r4,r3,r2,r1,r0*/  \
        *(volatile unsigned int *)(sp)=(unsigned int)(args);\
        (sp)=(sp)-4;/*cpsr*/         \
        *(volatile unsigned int *)(sp)=(unsigned int)(cpsr);\
    }while(0)

unsigned int get_cpsr(void){
    unsigned int p;
    asm volatile(
        "mrs %0,cpsr\n"
        :"=r"(p)
        :
    );
    return p;
}

int do_fork(int (*f)(void *),void *args){
    struct task_info *tsk,*tmp;
    if((tsk=copy_task_info(current))==(void *)0)
        return -1;
```

```
    tsk->sp=((unsigned int)(tsk)+TASK_SIZE);
    DO_INIT_SP(tsk->sp,f,args,0,0x1f&get_cpsr(),0);

    disable_schedule();
    tmp=current->next;
    current->next=tsk;
    tsk->next=tmp;
    enable_schedule();

    return 0;
}
```

在代码 9-6 中，do_fork 就是一个进程产生函数，它是完全依照图 9-5 的描述实现的。通过调用这个函数，程序可以轻轻松松地动态实现一个新进程，我们现在就来分析一下。

函数 copy_task_info 负责分配进程空间，为简化程序考虑，我们仅仅是从 0x30300000 地址开始，依次分配 4K 大小的区域来实现这段内存空间分配函数的。这对于我们这个示例操作系统来说不会造成什么影响，但却更加直观。而在实际的操作系统中，也只需使用页分配函数对此进行替换即可。

然后，函数通过宏定义 DO_INIT_SP 来实现堆栈寄存器的初始化以及进程函数的压栈工作。该宏压栈的顺序从高地址向低地址分别是进程入口地址、进程返回地址、R0 ~ R12 寄存器、函数参数、进程 CPSR，这些都是参考图 9-4 实现的。

对于一个新产生的进程来说，我们并不关心 R1 ~ R12 寄存器的具体值，但是也必须在堆栈中为这些寄存器留出空间来。而寄存器 R0 则负责为进程函数传递参数，因此程序需要将参数保存到 R0 寄存器所在的堆栈位置中。为了保证新进程的进程模式与调用 do_fork 函数的进程模式相同，我们通过 get_cpsr 函数获取上一进程的状态后，提取出后 5 位作为新进程的运行状态。

最后，为了能够让__common_schedule 函数成功返回新进程的 struct task_info 结构体，我们需要将系统内的所有进程结构体连接成链表。于是，do_fork 函数接下来的工作就是将新进程的 struct task_info 结构体添加到当前进程之后。在我们的操作系统中，我们设计了一套单向链表来连接进程。

链表本身属于基本的数据结构范畴，其操作方法非常简单。但不寻常的地方在于，在操作链表之前，我们调用 disable_schedule 宏临时关掉了进程

切换，而完成链表操作后，又调用 enable_schedule 宏恢复进程切换。这其实是一个与进程并发和同步有关的话题，不在本书的讲授内容之中。简单地说，这样做的目的是防止进程链表操作过程正在进行时，进程发生切换带来未知的后果。

这样，函数 do_fork 就完成了产生新进程的所有工作。读者朋友们应该还会对 current 这个变量心存疑惑。这里为了代码编写方便，我们将 current 定义成代码 9-2 的 current_task_info 函数，这种方法来源于 Linux。完整的代码如下所示：

代码 9-7

```
#define TASK_SIZE   4096

struct task_info *current_task_info(void){
    register unsigned long sp asm ("sp");
    return (struct task_info *)(sp&~(TASK_SIZE-1));
}

#define current current_task_info()
```

9.2.4 多个进程同时运行

能够让多个进程同时运行是一件非常激动人心的事情，我们马上就可以实现。

首先，我们需要用代码 9-3 的内容来改写“abnormal.s”文件的内容。

然后，将代码 9-4、9-5、9-6、9-7 以及下面这段代码 9-8 的内容保存成一个新的文件，名为“proc.c”。

代码 9-8

```
int task_init(void){
    current->next=current;
    return 0;
}
```

之后我们还要修改“boot.c”文件，删除掉 plat_boot 函数，并将代码 9-9 的内容添加进去。

代码 9-9

```
int do_fork(int (*f)(void *),void *args);

void delay(void){
    volatile unsigned int time=0xffff;
    while(time--);
}

int test_process(void *p){
    while(1){
        delay();
        printk("test process %dth\n",(int)p);
    }
    return 0;
}

void plat_boot(void){
    int i;
    for(i=0;init[i];i++){
        init[i]();
    }
    init_sys_mmu();
    start_mmu();
    task_init();
    timer_init();

    init_page_map();
    kmalloc_init();

    ramdisk_driver_init();
    romfs_init();

    i=do_fork(test_process,(void *)0x1);
    i=do_fork(test_process,(void *)0x2);

    while(1){
        delay();
        printk("this is the original process\n");
    };
}
```

在代码 9-9 中，程序首先调用了 task_init 函数。这个函数的功能只有一

个，那就是初始化原始进程结构体的链表结构，让 next 成员指向自身。这样，调用 do_fork 函数时，新的进程结构才能成功被添加。而后，我们两次调用 do_fork 函数，开辟两个新进程。这两个进程将会执行同一个函数 test_process，该函数仅仅是循环以十进制整数的形式打印进程函数参数。为了能在运行时对这两个进程做区分，我们分别在调用时传递了两个不同的参数供其打印。此时，原始进程仍在运行，我们让它循环打印 " this is the original process\n " 这段字符串，可以看出，这段代码就是对前面封装的进程函数的一系列简单调用。

最后，修改 Makefile 中的 OBJS 变量，将“proc.o”添加到目标文件列表中。

这样，我们的操作系统就可以编译运行了。不出意外的话，这段程序的运行结果将会是如下情况：

```
arch: arm
cpu info: armv4, arm920t, 41009200, ff00fff0, 2
mach info: name s3c2410x, mach_init addr 0x426c70
uart_mod:0, desc_in:, desc_out:, converter:
SKYEYE: use arm920t mmu ops
Loaded RAM   ./leeos.bin
Loaded RAM   ./ram.img
start addr is set to 0x30000000 by exec file.
helloworld
this is the original process
test process 2th
test process 1th
this is the original process
test process 2th
test process 1th
this is the original process
test process 2th
test process 1th
……
```

可以看出，三个进程依次被调度的情况出现了，我们的操作系统正在以多进程的方式运行！

9.3 总结

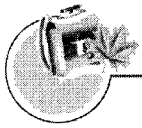
正如本章开头所讲的，进程永远都是操作系统的核心，这就是很多关于操作系统原理的书中都把进程作为开头的重点内容介绍的原因。

然而，本书的编排却恰恰相反，将进程的内容放在了最末尾。其中一个原因是，本书强调理论通俗易懂、重视实践易于操作。如果将进程的话题放到最前面，很多读者就有可能会在硬件体系结构、操作系统原理等知识都一知半解的情况下去编写大量的代码，实现一个最终可能都不能完全理解的进程调度程序。当我们一步一步、循序渐进地掌握了从硬件到算法等各种基础知识后，再来实现多进程，也就水到渠成了。

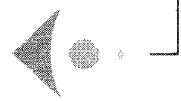
一个操作系统是否就此就结束了呢？远远还没有。

往操作系统核心看去，一旦程序以并行的方式去运行，那么进程调度算法的问题、进程之间的互斥和通信问题、受进程影响的内存和文件系统等问题都会蜂拥而至，即便再用这么长的篇幅，也很难一一说清楚。

这些内容再结合本书整整九章的描述，就构成了一套完整的操作系统理论。从这个角度来看，进程可以成为本书的结束，也可以成为新的知识的开始。



结束语



至此，本书的内容就全部结束了，感谢读者的支持，能够跟随书中的脚步一点点地走到这里！

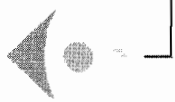
然而，本书虽然读完了，但这并不代表对操作系统的学习和研究就此终结。本书的结束标志着一个新的旅程的开始。

一方面，本书针对操作系统的结构虽然完整，但有很多细节我们并未涉及。这包含了嵌入式操作系统实时性问题、进程调度算法问题，进程间互斥和通信问题、多核问题、移植性问题，等等。本书只是为读者打开了一扇前往操作系统世界的大门。如果有可能，我也希望能够在另一本书里与大家再次见面，共同探讨那些未曾涉及的操作系统话题。

另一方面，读者虽然已经完成了属于自己的嵌入式操作系统，但这却仅仅是个开始。如果读者还有热情和精力，希望能够在如下两个方面尝试远行：尝试继续完善自己的操作系统，完善核心代码、增加操作系统健壮性、移植函数库、实现用户程序的标准化等；尝试将已有的操作系统代码移植到实际的硬件平台中。

最后，欢迎读者访问 www.leeos.org，就书里书外的一切话题与本人进行交流！

参考资料



[1]陈渝，谌卫军. 操作系统设计与实现 [M]. [美]Andrew S.Tanenbaum, Albert S. Woodhull. 第3版. 北京：电子工业出版社，2008：上册

[2]邵贝贝. 嵌入式实时操作系统 uC/OS II[M].[美]Jean J.Labrosse. 第2版. 北京：北京航空航天大学出版社，2003

[3]陈莉君，康华，张波. Linux 内核设计与实现[M].[美]Robert Love. 第2版. 北京：机械工业出版社，2006

[4]陈葆珏，王旭，柳纯录，冯雪山. UNIX 操作系统设计[M].[美]Maurice J. Bach. 北京：机械工业出版社，2004

[5]沈建华. ARM 嵌入式系统开发——软件设计与优化[M].[美]Andrew N.Sloss，[英]Dominic Symes，[美]Chris Wright. 北京：北京航空航天大学出版社，2005

[6]杜春雷. ARM 体系结构与编程[M]. 北京：清华大学出版社，2003

[7]邹恒明. 计算机的心智：操作系统之哲学原理[M]. 北京：机械工业出版社，2009

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

一步步写嵌入式操作系统

——ARM编

一次跟Android领军人物高焕堂先生聊天时，他的国外先进的开发工具、平台和操作系统就好比是武器，进武器去打仗（做应用层开发），一旦有一天我们跟外国人打起来，人家拿走我们的武器，我们就真的是一筹莫展了。

这句话很有道理，中国计算机技术整体水平的提高需要以大量自主研发的开发工具、平台架构以及操作系统为基础。不过，目前我们离这样的一个目标还相去甚远。

本书强调实践，力求能够帮助读者编写出属于自己的嵌入式操作系统。如果读者以本书为基础（或者哪怕从中得到了一丝灵感）开发出一些优秀的嵌入式操作系统，那将会是非常令人高兴的事情！

本书涵盖内容：

- **搭建工作环境**
  - 选择合适的开发环境
  - 开发工具的使用
  - ……
- **基础知识**
  - 使用C 语言写第一段程序
  - 用脚本链接目标文件
  - ……
- **操作系统的启动**
  - 启动流程
  - MMU
  - ……
- **打印函数**
  - 打印函数实例
  - 实现自己的打印函数
- **中断处理**
  - ARM 的中断
  - 简单的中断处理实例
  - ……
- **动态内存管理**
  - 伙伴算法
  - slab
  - ……
- **框架**
  - 驱动程序框架
  - 文件系统框架
- **运行用户程序**
  - 二进制程序的运行方法
  - 可执行文件格式
  - ……
- **进程**
  - 进程的实现原理
  - 进程的实现

上架建议：计算机>嵌入式/ARM

ISBN 978-7-121-12240-8

9 787121 122408 >

定价：39.00元

策划编辑：袁金敏
责任编辑：贾　莉
封面设计：李　玲